Vectorizing for Wider Vector Units in a HW/SW Co-designed Environment

Vectorizing for Wider Vector Units in a HW/SW Co-designed Environment

Rakesh Kumar[1], Alejandro Martínez[2], Antonio González[1, 2]
[1] Department of Computer Architecture, Universitat Politècnica de Catalunya, Barcelona, Spain
[2] Intel Barcelona Research Center, Intel Labs, Barcelona, Spain
rkumar@ac.upc.edu, alejandro.martinez@intel.com, antonio.gonzalez@intel.com

***Abstract*—SIMD accelerators provide an energy efficient way of improving the computational power in modern microprocessors. Due to their hardware simplicity, these accelerators have evolved in terms of width from 64-bit vectors in Intel´s MMX to 512-bit wide vector units in Intel´s Xeon Phi. Although SIMD accelerators are simple in terms of hardware design, code generation for them has always been a challenge. This paper explores the scalability of SIMD accelerators from the code generation point of view. We explore the potential problems in vectorization at higher vector lengths. Furthermore, we propose Variable Length Vectorization and Selective Writing in a HW/SW co-designed environment to get around these problems. We evaluate our proposals using a set of SPECFP2006 and Physicsbench applications. Our experimental results show an average dynamic instruction elimination of 33% and 40% and an average speed up of 15% and 10% for SPECFP2006 and Physicsbench respectively, for 512-bit vector length, over the scalar baseline code.**

Keywords - HW/SW Co-designed processor, Vectorization, Speculation, Dynamic optimization

I. INTRODUCTION

The last decade has seen emergence of Hardware/Software (HW/SW) co-designed processors as a solution to the power and complexity problems of modern microprocessors [4][5][10]. In order to reduce the power consumption and complexity, these processors incorporate simple hardware. Moreover, several dynamic optimizations are applied to achieve better performance.

Single Instruction Multiple Data (SIMD) accelerators provide higher FLOPs in an energy, area and complexity efficient manner. Due to their hardware simplicity, SIMD accelerators grow in size with each new generation. For example, Intel´s MMX [2] had vector length of 64-bits, whereas Intel´s recent SIMD extensions AVX [2] and Intel´s Xeon Phi [3] support 256-bit and 512-bit vector operations respectively. Although SIMD accelerators are amenable to scaling from the hardware point of view, generating efficient code at higher vector lengths is not straightforward. There are applications for which compilers just need to unroll loops with a higher unroll factor to fill the wider vector paths. However, there is another category of applications that does not have enough parallelism for vectorization at higher vector lengths. Generating code for these applications for wider vector units becomes a challenge.

In this paper, we explore the scalability of SIMD accelerators from the code generation point of view. We propose a speculative dynamic vectorization algorithm that can be implemented in the software layer of a co-designed processor. The proposed algorithm speculatively reorders and vectorizes memory operations. Moreover, we show that there are two key factors that thwart the performance at higher vector lengths: reduced dynamic instruction stream coverage for vectorization and huge number of permutation instructions. We propose Variable Length Vectorization and Selective Writing to tackle these problems. Our experimental results show average dynamic instruction elimination of 33% and speed up of 15% for SPECFP2006, for 512-bit vector length.

The main contributions of this paper can be summarized as:

- Identifies the bottleneck in vector code generation for wider vector units.
- Proposes Variable Length Vectorization to increase the dynamic instruction stream coverage.
- Proposes Selective Writing to reduce the number of permutation instructions.
- Evaluates the proposals using a set of SPECFP2006 and Physicsbench.

The rest of the paper is organized as follows: Section II provides motivation for the work presented in this paper and identifies key issues in efficient vector code generation for higher vector lengths. Section III describes the speculative dynamic vectorization algorithm. Section IV and V explain the proposed Variable Length Vectorization and Selective Writing techniques, respectively. Evaluation of the proposals using a set of SPECFP2006 and Physicsbench applications is presented in Section VI. Section VII presents related work and Section VIII concludes.

II. MOTIVATION

The trends in the recent past have shown that vector lengths are going to increase in the future microprocessors, e.g. Intel´s Xeon Phi. However, it is a challenge to generate efficient code to utilize these wider vector units. To demonstrate this fact, we vectorized floating point

instructions in SPECFP2006 for three different vector lengths of 128, 256 and 512-bits using the algorithm described in Section III. At a given vector length, all the vector instructions operate only on the maximum vector length and not on a subset of it. For example, for 512-bit vector length case, all the vector instructions operate on whole 512-bits and there is no vector instruction that operates only on 256 or 128-bits. Our results show that there are mainly two problems in vector code generation at higher vector lengths:

A. Reduced Dynamic Instruction Stream Coverage

We define dynamic instruction stream coverage as the number of dynamic scalar instructions vectorized. Figure 1 shows the dynamic instruction stream coverage for vectorization at different vector lengths normalized to the 128-bit case. Best, worst and average cases are shown. We divide the applications in two categories: First category applications have maximum dynamic instruction stream coverage at all the vector lengths, like 454.calculix. On the other hand, there are applications like 444.namd where dynamic instruction steam coverage falls a lot at vector length of 512-bits.

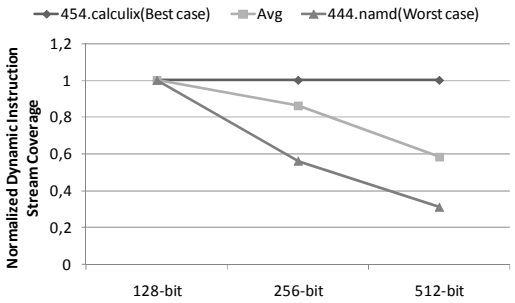

Figure 1. Dynamic FP Instruction Stream coverage for vectorization at 128, 256 and 512-bit vector lengths

If an application spends most of its time in loops with higher trip counts, it will benefit from higher vector lengths, since wider vector paths can be filled by unrolling the loops more number of times. However, as shown by the average case of Figure 1, this is not the case for most of the applications. We see a significant reduction in coverage as we increase the vector length.

B. Number of Permutation Instructions

When the input operands of a vector instruction are not available in a single vector register or are not in the same order as required by the vector instruction, permutation instructions are needed to arrange them in the correct order. Our results show that the number of permutation instructions grows significantly at higher vector lengths.

Figure 2 shows the number of the permutation instructions generated per vector instruction in SPECFP2006 normalized to the 128-bit case. As the figure shows, if we generate one permutation instruction for each vector instruction at 128-bit vector length, this number goes as high as 10 at 512-bit vectors in case of 444.namd. Also, there are applications for which this number does not grow that rapidly. However, the average behavior suggests that number of permutation instructions is going to be a problem at higher vector lengths.

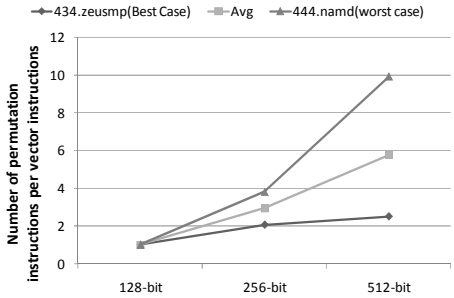

Figure 2. Normalized Number of Permutation Instructions generated per vector instruction

Both of these factors become a limitation as vector paths become wider and instead of performance improvements, it starts degrading compared to the lower vector lengths. This paper investigates both the problems and proposes Variable Length Vectorization and Selective Writing to solve the problems of reduced coverage and permutation instructions, respectively.

III. VECTORIZATION ALGORITHM

The software layer of our co-designed processor is called Translation Optimization Layer (TOL). TOL operates in three translation modes for generating host code from guest x86 code: Interpretation Mode (IM), Basic Block Translation Mode (BBM) and Superblock Translation Mode (SBM). Vectorization is done in SBM, which is the most aggressive optimization mode, after applying several standard compiler optimizations.

TOL starts by interpreting guest x86 instruction stream in IM and then progressively promoting to BBM and SBM as certain basic blocks become hotter. A superblock generally includes multiple basic blocks following the biased direction of branches. Loop unrolling plays a major roll in vectorization. Compilers unroll the loops a particular number of times to get sufficient independent instructions to fill the vector path. It is relatively simple to determine the unroll factor for loops with static trip count. However, for the loops, where the number of iterations are not known statically, it is difficult to decide the unroll factor. The availability of dynamic application behavior in HW/SW co-designed processors allows us to detect the loop unroll factor dynamically. Currently, we unroll loops with a single basic block, as the loops with no or minimum control flow provide maximum benefit [8].

B. Vectorization

The vectorizer packs together a number of independent scalar instructions, which perform the same operation, and replaces them with one vector instruction. The number of

scalar instructions packed depends on the data-types of scalar instructions and the vector length. Before describing the algorithm, we define a set of conditions that a pair of instructions must satisfy to be included in the same pack:

- The instructions must be performing the same operation.
- The instructions must be independent.
- The instructions must not have been included in another pack.
- If the instructions are load/store, they must be consecutive.

Vectorization starts by marking all the instructions that are candidates for vectorization. Moreover, we mark First Load and Store instructions. First Load/Store instructions are those for which there are no other loads/stores from/to adjacently previous memory locations. For example, if there is a 64-bit load instruction I_L that loads from a memory location [M] and there is no 64-bit load instruction that loads from address [M – 8], we call I_L First Load.

Vectorization begins by packing consecutive stores, starting from a First Store. The decision of starting with stores instead of loads is based on the observation that a given kind of operation always has the same number of predecessors, whereas the number of successors may vary depending on how many instructions consume the result. Consequently, following a bottom-up approach results in a more structured tree traversal than a top-down approach.

Once a pack of stores is created, their predecessors are packed, before packing other stores, if they satisfy the packing conditions. Moreover, if the last store in the pack has a next adjacent store, it is marked as First Store so that a new pack can start from it. Once all the stores are packed and their predecessor/successors chains have been followed, we check for remaining load instructions that satisfy the packing conditions and pack them the same way as stores.

Vectorization starting from adjacent loads/stores has an obvious limitation: if a superblock does not have any consecutive loads/stores, nothing can be vectorized. To tackle this problem, after packing all loads/stores and their predecessors/successors, we check if still there are some arithmetic instructions which can be packed together. If yes, we vectorize them and follow their predecessors/successors.

While traversing the predecessor/successor tree, if we find out that the predecessors of a pack cannot be vectorized, a permutation (Pack) instruction is generated. This Pack instruction collects the results of all the predecessors into a single vector register and feeds the current pack. Similarly if all the successors of a pack cannot be vectorized, an Unpack instruction is generated. This Unpack instruction distributes the result of the pack to the scalar successor instructions.

IV. VARIABLE LENGTH VECTORIZATION

Vector instructions in the current architectures, generally, operate on all the elements of the source vector registers and write the whole destination register. Therefore, compliers generate a vector instruction only when there are sufficient numbers of independent operations to fill the vector path, otherwise all the instructions are left in the scalar form. In the future microprocessor with wider vector paths it will lead to a lot of, otherwise vectorizable, code being left unvectorized. We propose Variable Length Vectorization (VLV) using masked vector instructions to vectorize the scalar code when it is not possible to fill the vector path entirely.

A. Code Generation

We modify our baseline algorithm of Section III to generate vector code with variable vector length. The modified algorithm starts by vectorizing for the maximum vector length. Once all the possible packs for the maximum vector length have been created, vectorizer reduces the logical vector length iteratively. At lower logical vector lengths, packs are created with smaller number of scalar instructions than required to fill the vector path. The left out positions are considered as no operations (nops).

Since the number of operations in the vector instructions varies depending on the logical vector length; we need to notify the hardware which vector lanes to enable and which ones not. We make use of mask registers for this purpose. The mask registers have one bit per vector lane. The bits containing ones signify the corresponding lanes are to be enabled; 0 means otherwise. We include the mask register in the instruction encoding in addition to the source and destination registers.

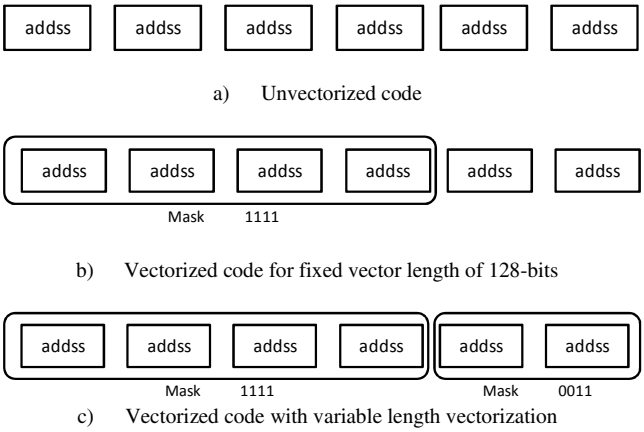

Figure 3 Variable Length Vectorization Example

Figure 3a shows unvectorized code having six independent single precision addition instructions. For a vector length of 128-bits, we can pack a maximum of four additions in a single vector instruction. The algorithm first packs four of the six instructions in a vector instruction and assigns a mask register with all ones to enable all the vector lanes, Figure 3b. A fixed vector length vectorization algorithm will stop at this point, since there are just two "addss" instructions left and at least four are required to generate a vector instruction. However, VLV algorithm continues and packs the remaining two addition instructions,

Figure 3c. Also, a mask register with ones only at lowest two positions is assigned to this instruction. It makes sure that only the two lower vector lanes are enabled during execution.

B. Hardware Requirements

From the hardware perspective, we do not really need to have real mask registers in the hardware. Since we need to enable only consecutive lower order vector lanes, the number of lanes to be activated can directly be encoded in the instructions encoding. This also saves the extra instructions, otherwise, needed to write the mask registers.

For the execution of a vector instruction, the hardware now reads not only the source registers but also a mask to enable only the required vector lanes. For example, for the vector instruction with mask "0011", Figure 3c, only two of the four vector lanes are to be activated. This is also important from the power consumption point of view, not to activate all the vector lanes for all the vector instructions.

It is important to note that the traditional vector processors support variable vector length through a vector length register. It needs to be set to the desired vector length before executing vector instructions. However, it is not the optimal solution for the processors targeting general purpose applications, where the vector length needs to be changed frequently. In this scenario, overhead of writing the vector length register would affect the performance severely.

V. SELECTIVE WRITING

In this section, first we present a technique to eliminate permutation instructions completely if the result of an instruction is read only by one instruction. Then, we present another technique to reduce the number of instructions required to pack N values, from N different registers, from N-1 to N/2.

A. Eliminating Permutations using Selective Writing

If the producer instructions of a vector instruction can not be vectorized, the results of these instructions have to be packed together before feeding the vector instruction. This is because the scalar producer instructions write their results to the lowest element of different vector registers, whereas the vector instruction needs them to be in a single register.

```
I0  addss   xmm0, xmm6
I1  addss   xmm1, xmm6
I2  mulss   xmm2, xmm7
I3  mulss   xmm3, xmm7
I4  shufps  xmm1, xmm0, imm
I5  shufps  xmm3, xmm2, imm
I6  blendps xmm3, xmm1, imm
I7  addps   xmm3, [M]
```

a) Traditional code sequence

```
I0  addss   vr4, vr0, vr6, imm
I1  addss   vr4, vr1, vr6, imm
I2  mulss   vr4, vr2, vr7, imm
I3  mulss   vr4, vr3, vr7, imm
I4  addps   vr5, vr4, [M]
```

b) Proposed code sequence

Figure 4 Packing scalar instruction results for feeding a vector instruction

Figure 4a shows a situation where producers of I7 (I0-I3) are not vectorized and their results are packed using a permutation instruction sequence (I4-I6). I0 to I3 write their results to the lowest elements of different vector registers. Then a sequence of three instructions, I4 to I6, packs these results in a single vector register xmm3, before feeding it to the vector instruction I7.

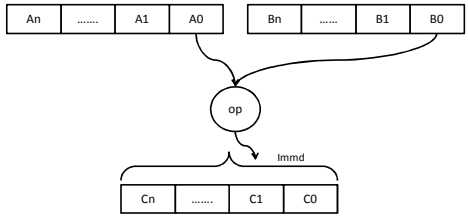

Figure 5 Functionality of the proposed arithmetic scalar instructions

We propose to modify the scalar instruction set so that the scalar instructions can selectively write in the different elements of a vector register, in the order they are needed by the vector instruction, Figure 5. This kind of selective writing capability is already available in the memory access instruction set of current architectures. For example, INSERTPS in Intel´s SSE can be used to write a 32-bit value loaded from memory to any part of the destination register. We extend it to arithmetic instruction set as well.

The new scalar arithmetic instructions, in addition to carry source and destination register numbers, also carry an immediate that specifies to which element of the destination vector register, the scalar result is to be written. If scalar instructions have written their results to a single vector register in the order in which they are needed by the vector instruction, the instruction sequence for packing these results is not needed anymore as shown in Figure 4b.

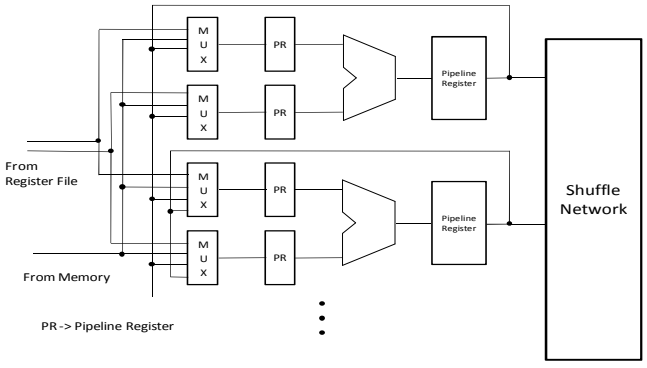

Figure 6 Operand Forwarding before shuffle

This scheme works as long as the result of a scalar instruction is consumed only by one instruction. In the case of more than one consumer, we would not get the maximum benefit out of this scheme. However, our analysis of SPECFP2006 shows that more than 70% of dynamic instructions have only one consumer.

The proposed scalar instructions can be viewed as an arithmetic operation followed by a shuffle. However, this does not affect the latency of these instructions, since the

results can be forwarded as soon as the arithmetic operation is finished. As Figure 6 shows, it requires only an additional input to the multiplexers (selecting input operands of the ALUs) from the output of the first vector lane (which performs scalar operations). Consequently, forwarding the results of the first vector lane to any other vector lane provides the functionality of a shuffle operation.

B. Reducing Permutation Instructions to Pack N values

A typical instruction sequence to bring 4 values from different vector registers to single vector register in x86 architecture is shown in Figure 7a. The first two shuffle instructions bring values selected by the immediate into register xmm1 and xmm3, respectively. Then a blendps instruction is used to combine the results from xmm1 and xmm3 into xmm3.

```
I0  shufps   xmm1, xmm0, imm
I1  shufps   xmm3, xmm2, imm
I2  blendps  xmm3, xmm1, imm
```

a) x86 instruction sequence

```
I0  Packps   vr6, vr0, vr1, imm
I1  Packps   vr6, vr2, vr3, imm
```

b) Proposed code sequence

Figure 7 Instruction sequence for packing 4 values from different registers into a single register

One of the main factors which force this instruction count to be N-1 is that these instructions write to all the elements of the destination register. If it is possible to write only the selective elements of the destination register, then this number can be brought down. In this case, the number of instructions required will depend upon the total number of different registers to be read and the number of registers that can be read by a single permutation instruction. In a case where we need to read N registers and the permutation instruction can read only two registers, we would need N/2 instructions. Moreover, we need a mechanism to tell which elements of the source registers are to be read and which elements of the destination register are to be written.

Figure 8 shows the functionality of the proposed permutation instruction. The proposed instruction has two input registers and a 16-bit immediate that tells the source and destination registers elements to be accessed. The first four bits of the immediate [0:3] tells which element of the first source register is to be read and the next four bits [4:7] tell where it is to be written in the destination. Similarly, bits [8:11] tell which element of the second source register is to be written to the destination element selected by the bits [12:15]. Note that Packps is similar to shufps with more freedom in choosing source element for each destination element. Therefore, their latencies will be similar.

The instruction sequence for replacing x86 instruction sequence of Figure 7a is shown in Figure 7b. In this case we are able to reduce the number of instructions required to two. For higher vector lengths, where we need to get 8 and 16 values in a register, we need just 4 and 8 instructions, respectively, instead of 7 and 15 instructions required by the original sequence. The down side of this scheme is that it requires N/2 instructions even if the values to be collected are in less that N number of registers. However, our experiments show that in SPECFP2006, on average, about 84% and 50% of permutations, for 256-bit and 512-bit vectors respectively, need to read N or N-1 registers to pack N values.

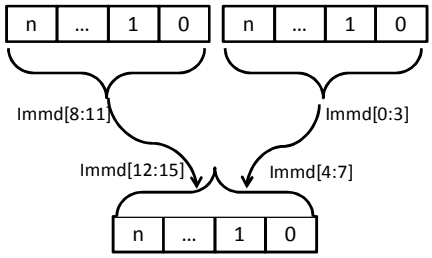

Figure 8 Functionality of the proposed Pack instruction

Discussion:

VLV and SW are better suited to HW/SW co-designed processors than traditional microprocessors mainly due to two reasons: First, they require significant ISA changes, since all the vector instructions now carry a mask register and the scalar instructions carry an immediate. It can be achieved in HW/SW co-designed processors transparently to the user/compilers but not in the traditional microprocessors. Second, the VLV algorithm is fairly simple to extend to compilers for the static trip count loops, however for loops with unknown trip count at compile time it becomes tricky. For fixed vector length, compiler can vectorize such loops by unrolling them enough times to fill the vector path and putting a run time check before the vectorized version to decide whether to execute it or not. However, for variable length vectorization, choosing a single unroll factor becomes difficult at compile time. The run time information of program behavior in HW/SW co-designed processors makes it straightforward to choose the correct unroll factor.

VI. PERFORMANCE EVALUATION

To measure the success of our proposals, we use a set of applications from SPECFP2006 [1] and Physicsbench [15] benchmark suites. For SPECFP2006 we instrument the benchmarks, using PIN [7], to find the most frequently executing routines. Then we simulate one billion x86 instructions starting from these routines. The benchmarks in Physicsbench are executed till completion. To evaluate our proposals, we use DARCO [9], which is an infrastructure for evaluating HW/SW co-designed virtual machines. DARCO executes guest x86 binary on a PowerPC-like RISC host architecture. The benchmarks are compiled with gcc version 4.5.3, with optimization flags "-O3 –ffast -math -fomit-frame-pointer".

For our experiments, we extended the host architecture to supports vector sizes of 128, 256 and 512-bits. Moreover,

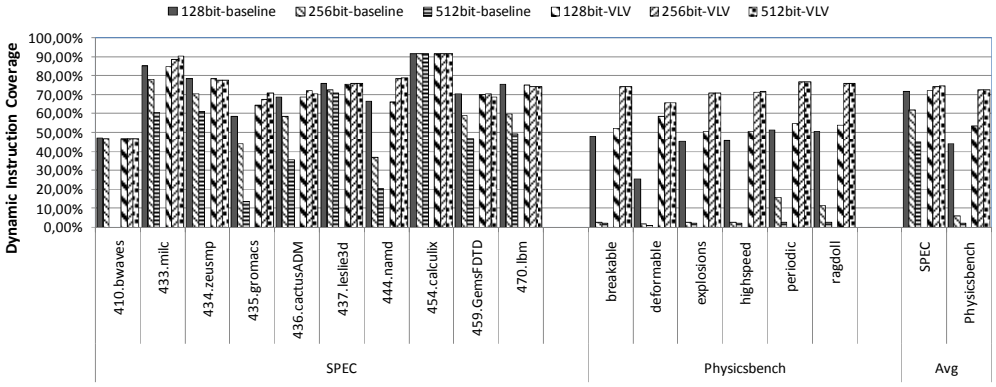

Figure 9 Dynamic Instruction Stream Coverage at Different Vector Lengths, baseline and VLV

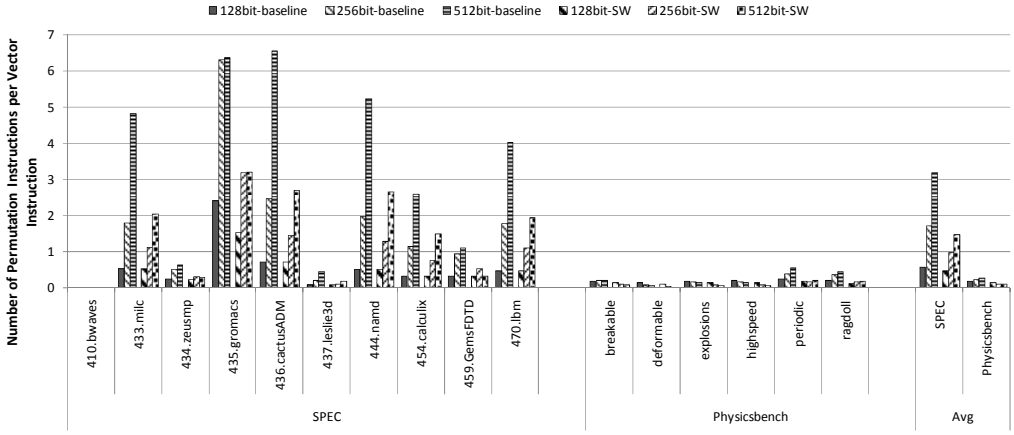

Figure 10 Number of Permutation Instructions per Vector Instruction, Baseline and SW

we consider only floating point operations for vectorization (because most SIMD optimizations tend to focus on them) and no integer operation is vectorized. For this reason, we show only the floating point instructions in the results presented. Table 1 shows the dynamic instructions eliminated, for a set of SPECFP2006 and Physicsbench, by our baseline algorithm of Section III and GCC (v4.5.3) for a vector length of 128-bits. On average our baseline vectorization algorithm outperforms GCC by 5% and 22% for SPECFP2006 and Physicsbench respectively.

Benchmark	TOL-baseline vectorization algorithm	GCC
437.leslie3d	30%	19%
436.cactusADM	10%	48%
ragdoll	30%	0%
deformable	6%	0%
SPECFP(avg)	16%	11%
Physicsbench(avg)	22%	0%

Table 1 Percentage of dynamic FP instructions eliminated by TOL and GCC vectorizers

A. Dynamic Instruction Stream Coverage

Figure 9 shows the dynamic instruction stream coverage for three vector lengths first without and then with Variable Length Vectorization (VLV). We will have maximum coverage when the number of instructions required to create a pack is minimum, i.e. two instructions. At 128-bit vector length the maximum number of 64-bit double precision operations that can be packed together is two. Therefore, 128-bit vector length provides maximum coverage, even without VLV, for double precision operations. Since all the SPECFP2006 benchmarks, except 435.gromacs, primarily operate on double precision floating point variables, they have maximum coverage at 128-bits as shown in Figure 9. For single precision floating point variables, Variable Length Vectorization helps increasing coverage even at 128-bit vector length, as is evident from the figure, for Physicsbench benchmarks and 435.gromacs.

For the vector lengths of 256-bit and 512-bits, the benchmarks can be divided into two categories. First, the benchmarks like 454.calculix and 437.leslie3d have maximum, or close to maximum, dynamic instruction stream coverage at higher vector lengths. The hottest loops of these benchmarks have enough iterations to fill the wider vector paths. Second, the benchmarks like 435.gromacs, 436.cactusADM, 444.namd and Physicsbench show drastic reduction in coverage as vector length increases, due to the lack of independent instructions to fill the wider paths. These benchmarks either have loops with less iteration or with complex control flow. For example, the hottest loops in

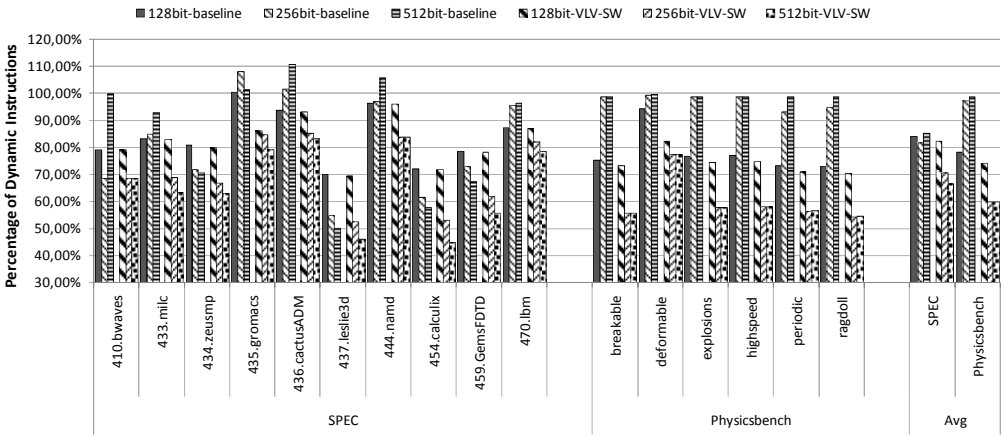

Figure 11 Percentage of Dynamic Instructions after Baseline and VLV-SW vectorization

410.bwave iterate four times, therefore, for 256-bit vector length it has the maximum coverage but for 512-bit, it drops down to zero. Benchmarks in Physicsbench have loops with really complex control flow and can not be unrolled. Using VLV, we bring the coverage for these benchmarks also to the maximum as shown in the Figure 9.

B. Permutation Reduction

Figure 10 shows the number of permutation instructions per vector instruction required at three vector lengths without and with Selective Writing (SW). Benchmarks like 434.zeusmp, 459.GemsFDTD and Physicsbench have, practically, the same amount of permutation instructions across all the vector lengths. Packing the instructions from the different iterations of unrolled loops avoids generation of permutation instructions in case of 434.zeusmp and 459.GemsFDTD. Physicsbench, however, has really small number of permutations since we fail to vectorize anything. On the contrary, 435.gromacs, 436.cactusADM and 444.namd show an increase in the permutation instructions at higher vector lengths. Complex control flow and lack of number of loop iterations forces us to vectorize straight line code which require higher number of permutation instructions. SW helps in eliminating significant number of permutation instructions for these benchmarks.

C. Putting Everything Together

Figure 11 shows the percentage of dynamic instructions after vectorization without and with VLV-SW. After applying both the optimizations all the applications perform better as vector length is increased. Applications like 433.milc, 435.gromacs, 436.cactusADM and Physicsbench which were earlier getting worse with increase in the vector length, compared to 128-bit vector length; now perform better. Overall, for SPECFP2006, vectorization with VLV-SW reduce unvectorized dynamic instruction stream by 17%, 29% and 33% for 128-bit, 256-bit and 512-bit vector lengths respectively. For Physicsbench we eliminate 40% more instructions compared to baseline vectorization and unvectorized code, at 256-bit and 512-bit vector lengths with VLV- SW.

The percentage of reduced instructions is same for 256-bit and 512-bit vector lengths in case of Physicsbench and 410.bwaves. The lack of availability of independent instructions at 512-bit vector length forces VLV to vectorize the code the same way as for 256-bit vector length. However, important point to notice is that we still have more instruction reduction than 128-bit case, which was not possible without VLV-SW.

D. Performance

We model a simple in-order processor, in congruence with the simple hardware design philosophy of the co-designed processors, with issue width of two. Microarchitectural parameters are shown in Table 2.

Parameter	Value
L1 I-cache	64KB, 4-way set associative, 64-byte line, 1 cycle hit, LRU
L1 D-cache	64KB, 4-way set associative, 64-byte line, 1 cycle hit, LRU
Unified L2 cache	512KB, 8-way set associative, 64-byte line, 16 cycle hit, LRU
Scalar Functional Units (latency)	2 simple int(1), 2 int mul/div (3/10) 2 simple FP(2), 2 FP mul/div (4/20)
Vector Functional Units (latency)	1 simple int(1), 1 int mul/div (3/10) 1 simple FP(2), 1 FP mul/div (4/20)
Registers	128-Integer, 128-Vector, 32-FP
Main memory Lat	128 Cycles

Table 2 Processor Microarchitectural Parameters

Figure 12 shows the percentage of execution time, at three vector lengths, after vectorization without and with VLV-SW. On average VLV-SW provide 12% and 15% over the unvectorized code, for vector length of 256-bit and 512-bit respectively, for SPECFP2006. Similarly, for Physicsbench, we get a speed up of 10% for with VLV-SW over unvectorized and baseline vectorization. There are several interesting points to note in Figure 12. First, even though we have higher dynamic instruction elimination, e.g. 33% for SPECFP 512-bit vector length, the speed up we get is smaller, 15% for SPECFP 512-bit vector length. This is

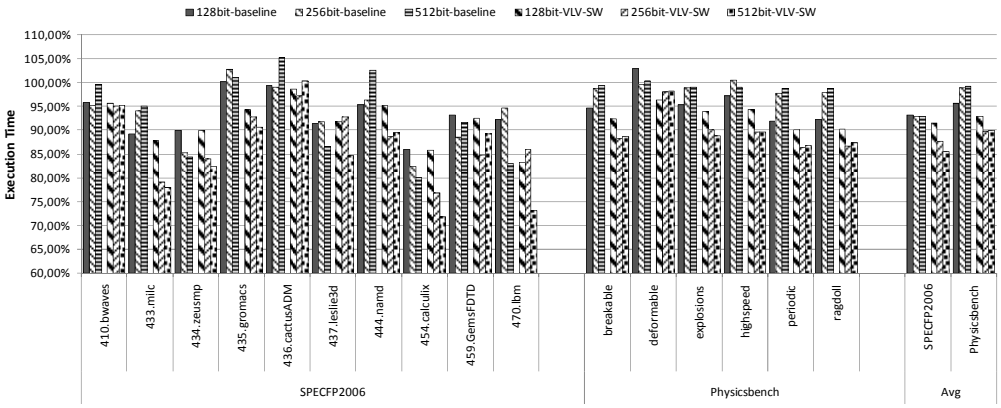

Figure 12 Execution Time for Baseline and VLV-SW Vectorized Code

because only 45% and 25% of dynamic instructions are floating point in SPECFP and Physicsbench respectively, which reduces the overall performance. Second, dynamic instruction reduction is more for Physicsbench, 40% compared to 33% of SPECFP2006 for 512-bit vector length; SPECFP shows more speed up, 15% compared to 10% of Physicsbench for 512-bit vector length. This is because Physicsbench has more integer instructions than SPECFP.

VII. Related Work

The proposal by M Woh et al. [14] for supporting multiple SIMD widths is the closest to our proposal of Variable Length Vectorization. They proposed a configurable SIMD datapath that can be configured to process wide vectors or multiple narrow vectors. Unfortunately, details of their vectorization algorithm for vectorization for multiple vector lengths are not provided.

Masked operations have been used in the past for vectorization of code with control flow [12][13]. However, we use them in the absence of control flow to increase dynamic instructions stream coverage. All the earlier proposals execute both if and else clauses and select the correct results based on the values in the mask registers. Our proposal, on the other hand, uses masked operations to increase the dynamic instruction stream coverage when there are not enough instructions to fill the wider vector paths. Significant amount of work has been done on the optimal generation of permutation instructions due to their obvious effect on performance [6][11]. However, previous work does not show effect of permutations at increasing vector lengths. These solutions focus on reducing the number of permutations required, whereas our solution reduces the number of instructions for each permutation.

VIII. Conclusion

This paper showed that widening the SIMD accelerators does not improve the performance for all the applications. We discovered two main problems hurting the performance of natural low vector length applications for wider SIMD units: Reduced dynamic instruction stream coverage and large number of permutation instructions. We propose Variable Length Vectorization to increase the instruction stream coverage and Selective Writing to reduce the number of permutation instructions.

Acknowledgments

This work is partially supported by the Generalitat de Catalunya under grant 2009SGR-1250, the Spanish Ministry of Education and Science under grant TIN 2010-18368, and Intel Corporation. Rakesh Kumar is supported by an FPI-UPC research grant.

References

[1] Standard Performance Evaluation Corporation. SPEC CPU2006 Benchmarks. URL http://www.spec.org/cpu2006/.

[2] Intel Corporation, Intel® 64 and IA-32 Architectures Software Developer´s Manual, Volume 1-3.

[3] "The Intel® Xeon Phi™ Coprocessor", : http://www.intel.com/content/www/us/en/high-performance-computing/ high-performance-xeon-phi-coprocessor-brief.html

[4] J. C. Dehnert et al. The transmeta code morphing™ software: using speculation, recovery, and adaptive retranslation to address real-life challenges. n Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization (CGO '03).

[5] K. Krewell. Transmeta Gets More Efficeon. *Micro-processor Report*, 17(10), 2003.

[6] A. Kudriavtsev et al. Generation of Permutations for SIMD Processors. In Proceedings of LCTES '05

[7] C. K. Luk et al. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05).

[8] S. S. Muchnick, Advanced Complier Design & Implementation, Morgan Kaufmann, 1997.

[9] D. Pavlou et al. DARCO: Infrastructure for Research on HW/SW co-designed Virtual Machines. In Proceedings of (AMAS-BT'11), held in conjuction with the ISCA-38 June 2011.

[10] S. S. Paul et al. BOA: Targeting multi-gigahertz with binary translation. In *Proc. of the Workshop on Binary Translation,* 1999.

[11] G. Ren et al. Optimizing Data Permutations for SIMD Devices. In Proceedings of PLDI '06. pages 118-131.

[12] J. Shin et al. Superword-Level Parallelism in the Presence of Control Flow. In Proceedings of the international symposium on Code generation and optimization (CGO '05), pages 165-175.

[13] J. Smith et al. Vector Instruction Set Support for Conditional Operations. In Proceedings of the 27th annual international symposium on Computer architecture (ISCA '00), pages 260-269.

[14] M. Woh et al. AnySP: Anytime Anywhere Anyway Signal Processing, In Proceedings ISCA '09, pages 128-139.

[15] T. Y. Yeh et al. Parallax: An architecture for real-time physics. In Proceedings ISCA '07, pages 232-243.